The TOM Programming Language

Pieter J. Schoenmakers
<tiggr@ics.ele.tue.nl>

DRAFT

as of January 9, 1998
for TOM version 1.0

Contents

Preface

This book provides an introduction to the TOM programming language, currently documenting version 1.0.

TOM was developed by Pieter Schoenmakers. The author can be reached at <`tiggr@ics.ele.tue.nl`>, or at

Pieter J. Schoenmakers
Eindhoven University of Technology
Room EH 9.26
P.O.Box 513
NL-5600 MB Eindhoven
the Netherlands

More information on TOM is available from the author or from the TOM WWW site at `http://tom.ics.ele.tue.nl:8080/doc/`.

Copying

This book—*The* TOM *Programming Language*—, the language it describes, and the tools discussed, (collectively denoted 'software' in the following) are *free*; this means that everyone is free to use them and free to redistribute them on a free basis.

Specifically, we want to make sure that you have the right to give away copies of the software that relate to TOM, that you receive source code or else can get it if you want it, that you can change the software or use pieces of them in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of the TOM related software, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for the software that relates to TOM. If the software is modified by someone else and passed on, we want their recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

The precise conditions of the licenses for the software currently being distributed that relates to TOM are found in the General Public Licenses that accompany them.

Chapter 1

Introduction

TOM is an object oriented programming language. A programming language can be used to instruct a computer. In an object oriented programming language the instructions consist of a description of objects and how they interact with each other.

An *object* is an entity with a certain *behaviour*. Objects can send messages to each other to evoke their behaviour. An object can maintain *state* to support its behaviour. If, for example, a `Counter` object is to respond as expected to repeated questions about its `nextValue`, it needs to maintain a notion of its current value. This current value would be the state of the `Counter` object.

As an example, sending the message `nextValue` to an object known as `counter` is written like this:

```
[counter nextValue];
```

A program can employ more than one object of the same kind, e.g. two `Counter` objects. The behaviour exposed by these objects is identical—they understand the same messages—but they maintain separate state: one `Counter` object in general has a different current value than another `Counter` object.

The `Counter` objects are said to be *instances* of the `Counter` *class*. The class describes the behaviour of its instances. Since all `Counter` objects are instances of the same class, they behave identically.

The `Counter` class itself is also available as an object. The most prominent behaviour of such a *class object* is its ability to create new instances.

> The fact that each class has a class object suggests the existence of an object describing the behaviour of the class object. Such an object exists; it is called the *meta class object*. To avoid inifinite meta levels, all meta class objects have a certain meta class object as their meta-meta class object. This implies that one of the meta class objects describes its own behaviour. However true, this observation does not lead to any spectacular possibilities—the language is about instances and their classes mostly; meta class objects aren't of any special interest.

Now the basic terminology of objects, classes and instances has been introduced, time has come to look at an actual TOM program.

1.1 Getting started

An important program in any programming language is the famous 'Hello, world!' It serves as a confidence test for the language installation at your site and shows you what to do to compile and run programs. This is what 'Hello, world!' looks like when written in TOM (the line numbers are for clarity; they are not part of the actual source):

```
 1 implementation class HelloWorld
 2
 3 int
 4   main Array argv
 5 {
 6   [[[stdio out] print "Hello, world!"] nl];
 7   return 0;
 8 }
 9
10 end;
11
12 implementation instance HelloWorld end;
```

The first line denotes the start of the implementation of a class named `HelloWorld`. The class ends with the `end;` at line 10. Between these lines a *method* is defined.

A method is a piece of code associated with an object. In this case the object is the `HelloWorld` class object. The method we define is called `main`. It returns a value of type `int` and accepts one argument of type `Array`. The name of the argument is `argv`; the code within the method can refer to the value of the actual argument through the name `argv`.

Line 6 contains three nested bracketed expressions; each expression *sends a message* to an object. The inner expression is `[stdio out]`. Here, the argumentless `out` message is sent to the `stdio` class object; this object is the *receiver* of the message. In response to this message, the corresponding method will be invoked, in this case the `out` method of the `stdio` class. Because of this correspondence, the terms 'sending a message' and 'invoking a method' are synonyms and they will be used as such throughout this book.

The `out` method of the `stdio` class returns an `OutputStream` object to which information can be written. This stream is usually connected to the user's terminal. Execution of the `main` method will resume when the `out` method has returned.

If we call the result of the first method invocation `x`, the second expression becomes

```
[x print "Hello, world!"]
```

This sends to the object `x` the message `print` with a single argument, the string '`Hello, world!`'. The stream object will print the string on the user's terminal and return itself. The third method invocation thus becomes

```
[x nl]
```

In response to this, the stream object will flush any buffered output to the user's terminal and emit a new-line character. Like the `print` method, it returns itself. This value is ignored here.

In line 7, the `return 0` causes execution of the method to be terminated and the value 0 to be returned.

> The `main` *method* is special, in that in every program exactly one class object must provide a `main` method. Upon execution of the program, this `main` method is invoked. When this method returns, the program terminates. The value returned by the `main` method is available by the calling program. On many machines, a value of 0 indicates success; any other value is assumed to indicate failure.

In line 12 the instances of the `HelloWorld` class are descibed. Since we have no use for them, the definition can be empty.

1.2 From source to executable

A program in source form can be fun to read, but sources are rather useless if they can not be transformed into something executable. This section will detail the steps involved in turning TOM source into an executable. If you are not interested in the details, you can safely skip this and continue to read the next section (1.3, page 6), which explains how you'd normally compile TOM programs.

The first step is creating the source file containing our example program. You can enter the example from the previous section manually or simply copy copy it from `/usr/local/tom/ttpl/ex/hello.t`[1]. In any case, you should end up with a file `hello.t` in a preferrably empty directory.

The next thing to do is to run the interface generator, *gi*. Apart from its main task—generating interfaces; hence its name—it is also responsible for creating the so-called *unit file*. A *unit* is the largest building block in TOM. A program is a unit, and each library is also a unit. The unit file declares which classes are defined in every file contained in the unit.

We will elaborate on the interface generator later on. For now it is important to know that you must run it because it will generate the unit file. Accomplish this by entering the following command line[2]:

```
$ gi -u hello -U tom hello.t
```

(You don't actually enter the `$`. It is here to remind you that the line is a shell command.)

This command tells gi:

`-u hello`
: To create the unit file for the unit `hello`. The name of the unit file is the name of the unit, with the extension `.u`. Thus our example unit file will be called `hello.u`.

`-U tom`
: The unit `hello` depends on the unit `tom`. The `tom` unit contains, among different kinds of useful objects, such as the `stdio` class used in our example program, also the runtime library which is necessary for executing TOM programs.

---

[1] The actual filename can be different on your site. As with all filenames mentioned on these pages, contact your local wizard in case of problems.

[2] If you experience problems, such as a 'Command not found', the environment variable `PATH` does not contain the directory where `gi` resides. Ask you local wizard which directory that is, and add it to the `PATH`. When installed out of the box, `gi` resides in `/usr/local/tom/bin`.

hello.t
This is the single file in our example unit.

Now that all necessary files are in place, it is time to run the compiler, which will turn a TOM source file into a C source file. Enter the following command:

```
$ otmc hello
```

otmc is the TOM compiler. It will search the current directory for the unit file[3]. After it has verified that the unit file indeed contains the file **hello**, it will read **hello.t** and produce **hello.c**.

Next, we let the C compiler do its work. The code produced by **otmc** can only be compiled by the GNU C Compiler (gcc), since it employs gcc specific constructs.

```
$ gcc -c hello.c
```

This turns the file **hello.c** into **hello.o**, which is the object file corresponding to the TOM source file **hello.t**.

Normally, a unit contains several source files, and you must run both **otmc** and **gcc** for every file in the unit.

When all files have been compiled—only **hello.t** in our example—you'd expect time has come to link the object files, with any libraries needed, to create the executable. However, when building a TOM program, one extra step is needed before the linking: the resolver must be run to build the data structures needed by the runtime library.

```
$ otmr
```

Similar to **otmc**, **otmr** operates on the unit file it finds in the current directory, creating a C source file containing the datastructures. In our example, it will create **hello-r.c**. Obviously, this needs to be compiled too.

```
$ gcc -c hello-r.c
```

Now we're really done compiling and the objects can be linked together to form the executable[4].

[3] If the directory contains more than one unit file, **otmc** will complain. The unit file can be explicitly indicated by a **-u hello**.

[4] It is possible that some extra flags are needed when linking, again depending on your site. Ask your local wizard for details. For example, on Linux you'll need to add **-ldl**, on HP/UX add **-ldld**, and on NeXSTEP add nothing.

```
$ gcc -o hello hello.o hello-r.o /usr/local/tom/tom.o
```

Obviously, the objects `hello.o` and `hello-r.o` are linked against the TOM library object and any implict libraries included by `gcc`.

When you've successfully completed the above steps, you can now run Your First TOM Program.

```
$ ./hello
Hello, world!
```

1.3 Automated compilation

In the previous section, all commands in the process from source to executable were entered manually. Obviously, this is tedious. In addition it becomes error-prone when the number of source files making up the executable increases. Luckily, the `make` utility was invented long ago.

`make` operates on *makefiles*, usually named `Makefile`. A `Makefile` contains a description of the dependencies between the files in a project, and commands to recreate files when they have become out-of-date.

The GNU project, in their quest for a free UNIX, have also written a `make` utility. As with a lot of GNU software, GNU `make` offers superior functionality over the original. Therefore the TOM Project uses GNU `make`. To remind you of this, all makefiles are called `GNUmakefile`.

To build TOM programs, you normally use the TOM makefiles[5]. This is a collection of makefiles, one for each kind of project. The makefiles are usually installed in `/usr/local/tom/makefiles`. To use them for the 'Hello, world!' example, all you need to do is create a file `GNUmakefile`, with the following contents:

```
UNIT= hello
USES= tom

TOM_SRC= hello

include /usr/local/tome/makefiles/GNUmakefile.bin
```

When you have edited any of the source files, to regenerate the program just enter the following command line[6]:

[5] The TOM makefiles are probably installed on your system. If they are not, they are available from `http://tom.ics.ele.tue.nl:8080/distrib/makefiles.tar.gz`.

[6] Or the equivalent rune needed to invoke GNU make, such as possibly, `gmake` or `gnumake`.

```
$ make
```

All you need to remember when using the template `GNUmakefile` is that if you add or remove files from the `TSRC` macro, you must remove the unit file, so it can be recreated.

Now you know how to create an executable from TOM source files, time has come to dive into the language itself, starting with the next chapter.

Chapter 2

Expressions

In the previous chapter you've seen how a TOM program is built from files, which contain classes and instances, which contain methods. This chapter explains the basics of methods, what you can make them do, and how to write that down in TOM.

Everything in a method body is an expression. We'll start with the simplest kind of expression: constants.

2.1 Numeric constants

An example of a rather simple expression is `1 + 2`. When executed its value is `3`, just like you'd expect. Both the `1` and `2` themselves are expressions too. They are the simplest of expressions, and called *constants*. As with all expressions, constants have a type. The type of `1` is `int`.

`int` is one of the numeric TOM types. Table 2.1 lists all numeric types available in TOM. If you're familiar with C, you'll recognize most of them. The difference is that in TOM the precision, range and signedness of the types are fully defined: there is no difference between machines. Furthermore, the `char` and `short` C types have been replaced by `byte` and `char` respectively, for reasons which will become apparent soon.

There are several kinds of numeric constants, each with a specific type. Constants like `1`, `123456`, `0377`, and `0xff` are of type `int`. A leading zero, as in `0377`, denotes a number in octal notation; `0xff` with its leading `0x` denotes a number in hexadecimal notation. The case of the hexadecimal digits is ignored, thus `0Xff`, `0xFf`, `0XFF`, and `0xff` are all equal.

Table 2.1: Numeric types

type	description
`byte`	8 bit unsigned integer
`char`	16 bit unsigned integer
`int`	32 bit signed integer
`long`	64 bit signed integer
`float`	single precision floating point
`double`	double precision floating point

An integer constant suffixed with `l` or `L` is of type `long`. Thus `0L` is a `0` typed `long`. If an integer constant, which is not an explicit `long`, is too large to be held by an `int`, its type will be `long`. If even a long can not hold the value, an error is issued by the compiler.

A `byte` constant is written as a character enclosed in single quote characters. Thus, `'a'` is a `byte` with a value of 97; 97 being the ASCII value of the letter 'a'[1]. The quote itself can be escaped using a '\' (backslash). Thus, `'\''` is a `byte` with a value of 39. To get a backslash, it too must be escaped—`'\\'`.

Not all `byte` values can be entered as a character constant, simply because not all ASCII values translate to printable (and typeable!) characters. Such characters can be entered by escaping their octal value. Thus, `'\041'` is the capital letter `A`.

In general, it is awkward to have to remember numeric values for often-used non-printable characters. Table 2.2 lists the shorthands of important character constants which stand for some of the unprintable ASCII values.

A `float` constant is a number which includes a decimal point, an exponent, or both. Thus `1.0`, `1e23`, and `4.2e1` are all floating point constants of type `float`.

Floating point constants of type `double` must have an exponent part and have 'd' as the exponent indicator. Thus `1d` is a floating point 1 of type `double`.

A floating point constant which seems to be a `float`, but the value of which is too large to be held by a `float`, is also taken to be a `double`. If

[1]Most of the Western world knows that ASCII and the 8 bit ISO character encodings have their limitations. In future versions of TOM, the character and string constants will deploy 16 bit Unicode characters. This is the reason for the 16 bit type to be called `char`. In addition, most of the non-Western world knows that even Unicode doesn't cover everything...

Table 2.2: Important non-printable characters

constant	value	name	description
`'\b'`	`0x08`	`BS`	backspace
`'\f'`	`0x0c`	`FF`	form feed
`'\n'`	`0x0a`	`NL`	new line
`'\r'`	`0x0d`	`CR`	carriage return
`'\t'`	`0x09`	`HT`	tab
`'\v'`	`0x0b`	`VT`	vertical tab

the value of a constant is too large to even fit a `double`, the compiler will issue an error.

2.2 Operators

Expressions are composed of operators and operands. Operators come in three flavours: with 1, 2, or 3 operands. Each operand in turn is an expression.

Each operator has a priority. For instance, the priority of the `*` is higher than that of `+`, causing `1 + 2 * 3` to be interpreted as first multiplying `2` by `3` followed by the addition of `1`. Parentheses can be used to impose a different evaluation order: `(1 + 2) * 3` will first add `1` and `2`, multiplying the result with `3`.

You can experiment with expressions by modifying the `hello.t` program to print something different from its familiar message. Here are a few examples; remember to run `make` to have the program rebuilt after you have modified the source.

```
[[[stdio out] print 1 + 2 * 3] nl];
[[[stdio out] print (1 + 2) * 3] nl];
```

And a more daring example—notice how `print` accepts more than one thing, grouped by parentheses and separated by commas.

```
[[[stdio out]
  print ("0 F = ", 5.0 / 9.0 * (0.0 - 32.0), " C")] nl];
```

Table 2.3 lists all operators. Operators nearer to the top have a higher priority than those below it. In the same group, between horizontal lines, operators have the same priority.

Table 2.3: Operators

operator	arity	associativity	description
`++`, `--`	1	right	increment, decrement
`-`, `~`, `!`	1	right	minus, inversion, not
`*`, `/`, `%`	2	left	multiply, divide, modulo
`+`, `-`	2	left	add, subtract
`<<`, `>>`	2	left	arithmetic shift
`>>>`	2	left	logic shift right
`&`	2	left	bitwise and
`|`	2	left	bitwise or
`^`	2	left	bitwise exclusive-or
`<`, `<=`, `>=`, `>`	2	left	ordered comparison
`==`, `!=`	2	left	equality comparison
`&&`	2	left	boolean and
`||`	2	left	boolean or
`->`	2	left	implies
`?:`	3	right	if-then-else
`=`, etc.	2	right	assignment (see text)

The unary minus returns the negation of its numeric argument. Thus, evaluating any of the expressions `-1`, `-(1)`, and `-(2 - 1)` all return the value -1. Negation preserves the type of its argument: negating an `int` value results in another `int` value.

The bitwise inversion operator, `~`, returns, given an integer numeric argument, an integer numeric value of the same type, but with all 0 bits replaced by 1 bits, and vice versa. Thus, `~0` returns -1.

The boolean not operator, `!`, returns `FALSE` iff its operand has the default value for its type, and `TRUE` otherwise. For example, the default value of numeric types is 0, so `!0` will return `TRUE` and `!456` will return FALSE.

The value returned by `!` has the `boolean` type. The only values of this type are `TRUE` and `FALSE`, also known as `YES` and `NO`[2].

Apart from `>>>` and `->` the binary operators perform the same function as their C, and many other languages, equivalents. `>>>` shifts a signed number in the same way as `>>` shifts an unsigned number in C. The implication operator, `->`, performs the boolean implication: `a -> b` is equivalent to `!a`

[2] The values `TRUE` and `FALSE` are not part of the language; they are predefined constants equivalent to `!0` and `!TRUE`, respectively.

Table 2.4: Examples of `>>>` and `->`

expression	result
`128 >> 1`	`64`
`-1 >> 1`	`-1`
`-1 >>> 1`	`0x7fffffff`
`TRUE -> FALSE`	`FALSE`
`TRUE -> TRUE`	`TRUE`
`FALSE -> TRUE`	`TRUE`
`FALSE -> FALSE`	`TRUE`

`|| b`. Table 2.4 shows a few examples of their use.

The operators `*`, `/`, `+`, `-`, `<`, `<=`, `>=`, and `>` operate on any numeric type. The `%`, `<<`, `>>`, `>>>`, `&`, `|`, and `^` operate on integer numeric types. `&&`, `||`, and `->` operate on the `boolean` type. Furthermore, `==` and `!=` operate on any type, including those to be introduced later on.

All binary operators are left associative, except the assignment operator[3], `=`, which is right-associative. The order of evaluation of the operands follows the associativity of the operator. Thus, `1 + 2 + 3` is interpreted as `(1 + 2) + 3`, and the `3` is only evaluated after the addition of `1` and `2`—not that it makes any difference in this simple case. Furthermore, `a = b = c` means `a = (b = c)`. (We will return to the assignment operator later on—they aren't much use when constants and operators are the only building blocks at hand.)

The boolean operators are *short-circuited*. This means that if enough information is known about the operands that the result of the whole expression is known the evaluation of any remaining operands is skipped. Thus, `FALSE && x` returns `FALSE` without ever evaluating `x`, since its value does not matter.

There is only one ternary operator, `?:`, also known as the if-then-else operator. For example, to compute the maximum of `a` and `b`, one could use `a > b ? a : b`, meaning that if `a` is larger than `b`, the second expression (in this case `a`) will be evaluated and returned as the result; otherwise the result of the third expression (`b`) is the result of the whole expression.

[3] The 'etc' in table 2.3 stand for all modifying assignment operators. These have the same priority and associativity as the assignment operator '`=`'. They are are `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `>>>=`, `&=`, `|=`, `^=`, `&&=`, and `||=`. For example, `a *= b` is identical to the assignment `a = a * b`.

In general, when used as `x ? y : z`, the type of `x` must be `boolean`, and the type of `y` must equal the type of `z`. Furthermore, only one of `y` and `z` will be evaluated.

2.3 Local variables

It is often necessary to keep the result of an expression for later use. A very good example of this is a counter: if you count from 1 to 10, when you're ready to increase the counter's value you must know what its current value is. A local variable can be used to hold such values. For example, if `a` is a variable, then the assignment

```
a = 1;
```

will assign to `a` the value `1`. When followed by the expression `a + 7`, the result of that expression will be `8`.

As you will have guessed, each variable has a type. Before a local variable can be used, it must be declared, not only to indicate its type but also to declare its existence. The variable `a` used above would be declared as:

```
int a;
```

You can experiment with variables by modifying the `main` method of our example program again.

```
int
  main Array argv
{
  int a = 11, b = 6;

  [[[stdio out] print ("a = ", a, " b = ", b)] nl];
  int c = a * b;
  [[[stdio out] print ("a * b = ", c)] nl];
  c = a + b;
  [[[stdio out] print ("a + b = ", c)] nl];

  return 0;
}
```

This example shows that multiple variables can be declared in one declaration, and that an assignment to a variable can be included in its declaration. In fact, when such an initial value is omitted, the variable will be set to

`0`. Verify this by omitting the initialization of `b` (i.e. the '`= 6`') from the example and running the program again after rebuilding.

In general, if a variable declaration does not specify an initial value, the value of the variable will be the default value of the variable's type, e.g. `0` for numeric types and `FALSE` when the type of the variable is `boolean`

2.4 Loops

One of the more valued features of computers is their ability to repeat, and repeat often. The *loop* is the crucial language construct underlying concept. Would you like to compute the conversion table from Celsius to Fahrenheit for every value between -100 and 100? Here's how to automate that.

```
int
  main Array argv
{
  float celsius = -100.0;

  while (celsius <= 100.0)
    {
      float fahrenheit = 32.0 + 9.0 / 5.0 * celsius;

      [[[stdio out] print (celsius, " ", fahrenheit)] nl];
      celsius = celsius + 1.0;
    }

  return 0;
}
```

The `while` loop tests its condition, in this case whether `celsius` still is less than `100.0`, and as long as the condition is true, the expression following it is executed. This expression is called the *body* of the loop. The loop condition is re-evaluated every time the body has been evaluated.

The example shows another noteworthy item: the body expression actually consists of a declaration and two expressions, separated by semicolons (`;`) and enclosed in braces, { and }. Such an expression sequence between braces is called a *compound expression*. The type and value of such a compound expression are the type and value of the last expression within the compound. This value is not always used, like in this example[4].

[4] The body of every method is enclosed in braces and thus a compound expression. The value of this expression is ignored however, since a `return` mechanism is used that provides much desired functionality.

Two other loop constructs are available, which basically are nothing more than a variation on the `while` loop theme.

The first variation is the `do while` loop. For example, the following code prints the numbers from `0` to `9` (inclusive or exclusive?).

```
int
  main Array argv
{
  int counter;

  do
    {
      [[[stdio out] print counter] nl];
      counter = counter + 1;
    } while (counter < 10);

  return 0;
}
```

The difference with the plain `while` loop is that the condition is evaluated *after* the body has been executed, i.e. the body is executed always at least once, whereas the body of the `while` loop can be skipped if the loop condition evaluates to `FALSE` before the body would be entered for the first round.

The third, and last, loop variation is the `for` loop. Like the `while` and `do while` loops, its syntactical origins stem from C. Here is a Fahrenheit to Celsius converter for degrees between -100 and 100, written down using a `for` loop.

```
int
  main Array argv
{
  float f;

  for (f = -100.0; f <= 100.0; f++)
    [[[stdio out]
      print (f, " ", (f - 32.0) * 5.0 / 9.0)] nl];

  return 0;
}
```

The `for` loop starts with three expressions, separated by semicolons. The first expression, in this case `f = -100.0`, is always executed. The second expression is the condition of the loop. If it evaluates to `TRUE`, the body is evaluated, followed by the evaluation of the third expression, `f++`, which increments `f` by 1. Then the condition is re-evaluated, and, if it is `TRUE`, the body, and so on.

As has been stated at the start of this chapter, everything in a method is an expression. Obviously then, loops also have a value. The value of a loop is the value of the body expression the last time it was executed. If the body is never executed, as can be the case with `while` and `for` loops, the value returned is the default value for the type.

2.5 More operators

The last loop example showed an operator which was not yet introduced, namely `++`. Its effect is to increment its operand, `f` in the example, by `1`. Similarly, `f--` would decrease `f` by `1`.

The increment and decrement operators can be used in a postfix notation, as in the example, or a prefix notation, as in `--f`. The difference between these notations is the value returned by the expression. The postfix notation returns the value of the variable before the modification; in prefix notation, the value of the expression the new value of the variable.

Given the fact that the value of a compound expression is the value of the last expression contained in the compound, `f++` is identical to

```
{
  int g = f;
  f = f + 1;
  g;
}
```

and `--f` equals

```
{
  f = f - 1;
}
```

The last kind of operators to be introduced are the modifying assignment operators. For example, `f = f + x` can be written as `f += x`. The same is true for every other binary operator. The precedence of these operators is equal to that of the normal assignment (they are the 'etc' in table 2.3).

2.6 Conditionals

Next to loops, conditional expressions are another kind of important language constructs. The `?:` operator introduced in section 2.2 is an example of a conditional expression: *if* the condition is `TRUE`, *then* execute what follows the `?`, *else* execute what follows the `:`.

This section introduces a semantically equivalent to the `?:` operator, but, as with a lot of TOM language constructs, with a syntax pleasing the eye accustomed to C.

The following example shows the use of the `if else` construct.

```
int
  main Array argv
{
  int n = [argv length];

  if (n == 0)
    [[[stdio out] print "no arguments"] nl];
  else
    [[[stdio out] print (n, " arguments")] nl];
}
```

If this program is invoked without any arguments, the `Array argv` will not contain any elements, and will return `0` when asked for its `length`. Consequently, the program's output will be `no arguments`. When invoked with at least one argument, it will report the number of elements in `argv`, i.e. the number of arguments.

When you have modified the `hello` program to have the above `main` method, the following is a example of its output when run.

```
$ ./hello
no arguments
$ ./hello tiny little program
3 arguments
$ ./hello "little robot"
1 arguments
```

And thus we discover a bug in our program: the output when invoked with a single argument is wrong since '1 arguments' is not proper English. You are hereby challenged to fix this, armed with the knowledge that what follows an `else` is an expression, just like the `if else` is an expression.

In contrast to the `?:` operator, the `else` branch of an `if else` conditional is optional. If the condition evaluates to `FALSE` and the `else` branch is missing, the value of the whole expression becomes the default value of the expected type.

For example, after the following expressions, the value of `a` will be `0`.

```
int b = 3;
int a = if (b < 0) b;
```

Chapter 3

Methods

Now the basics of what can be put in a method are clear, hopefully, we can move on to actually building new methods.

As was shown by the `main` method in the previous examples, a method is associated with a name, argument types, and a return type. The name can consist of more than a single part, in which case each part must be followed by an argument—remember how the `out` method of the `stdio` class does not need any arguments.

For instance, in the following example, the method's name is `multiply by`;

```
int
  multiply int a
        by int b
{
  return a * b;
}
```

This method accepts two arguments, each of type `int`, and returns another `int`. The body of the method simply returns the result of multiplying `a` and `b`.

Notice how every name part of the method starts on a new line, and that the nameparts are right justified. This is considered a readable style of writing method declarations. It is much more readable than written like this:

```
int multiply int a by int b { return a * b; }
```

To test our `multiply by` method we add it to the example program, between the start and the end of the class implementation, and test it using the following `main` method:

```
int
  main Array argv
{
  int i, n = [argv length], result = 1;

  for (i = 0; i < n; i++)
    {
      String s = [argv at i];
      result = [self multiply result by [s intValue]];
    }
  [[[stdio out] print result] nl];

  return 0;
}
```

The receiver of the `multiply by` message is `self`. `self` is an implicit argument to every method; it is the receiver of the message which caused invocation of the method. In this example, `self` is the `HelloWorld` class object.

This example also shows the use of a `String` object, which is retrieved from the array `argv`. An `Array` stores the objects it holds in a sequence, and using the `at` message, one can ask for the object at a certain index, as long as the index is within range, in this case $0 \leq$ `index` $< [argv\ \ length]$.

The full declaration of the `at` method is

```
Any
  at int index;
```

That is, an `Any` object is returned and since we know it is a `String`, we are allowed to assign the value returned to a `String`. In fact, as the name `Any` suggests, we can assign an `Any` to a variable of *any* class.

In the example, the `String s` is asked for its `intValue`. `intValue` is a `String` method which returns the integer value held by the string, in a straightforward way. For example, when asked for its `intValue`, the string `"123abc"` will return 123, `"abc"` will return 0, and `"0x123abc"` will return 1194684.

> It is customary for method name parts to use mixed case identifiers, starting with a lowercase letter. Class names also follow the mixed case convention, but they start with a capital letter. Variables should be all lower case, with the words within the identifier separated by '-' or '_'.

When recompiled, the `hello` program will, when run, output the result of multiplying its arguments.

```
$ ./hello 23 2
46
$ ./hello
1
$ ./hello 123456789 98
-786136566
```

The program behaves as expected, apart from the last example. Two positive numbers, when multiplied should return a positive number. However, when 123456789 is multiplied by 98 the answer, 12098765322, is too large to fit a signed 32 bits value, which is the `int` used for the `result` and the `multiply by` method. Obviously, the problem can be shifted to the 64 bits limit by using `long` values, but that is not a real solution; it is important that you realise the existence of such limits.

3.1 Tuples

There are occasions, when returning a single value from a method does not suffice, or when it is tedious to have multiple method name parts just to have multiple arguments. TOM solves both problems through tuples.

A *tuple* is a group of values within parentheses, separated by commas. For instance, `(1, 3.14)` is a tuple. The type of a tuple is the *tuple type* of which the elements are the types of the elements of the tuple. The type of the example is `(int, float)`.

An example of a method using tuples is the `substring` method from the `String` class.

```
String
  substring (int, int) (start, length);
```

This method has a single argument, the tuple `(start, length)`. As with a lot of other `String` methods, this tuple is used to select a range of elements, starting at index `start`, and running for `length` elements, where a `length` of `-1` means infinity.

The following example program, when run, will show a running text, probably best viewed on a slow terminal (of, say, 2400 baud).

```
int
  main Array argv
```

```
{
  String text = "Testing...  1 2 3", spaces = "          ";
  int len = [spaces length], finish = len + [text length];
  int num = (![argv length] ? 100 : [argv[0] intValue]);
  int i, count;

  for (count = 0; count < num; count++)
    for (i = 0; i < finish; i++)
      {
        if (i < len)
          {
            /* Case 1: Spaces followed by start of text.  */
            [out print [spaces substring (0, len - i)]];
            [out print [text substring (0, i)]];
          }
        else if (i < finish - len)
          {
            /* Case 2: Start and end the line in the text.  */
            [out print [text substring (i - len, len)]];
          }
        else
          {
            /* Case 3: Text followed by spaces.  */
            [out print [text substring (i - len, finish - i)]];
            [out print [spaces substring (0, len - (finish - i))]];
          }
        /* Go back to the start of the line.  */
        [out print '\r'];
      }

  = 0;
}
```

As an example of a method returning a tuple, let's look at using an `Enumerator`. An `Enumerator` can be used to traverse a collection of objects. After asking the collection for an enumerator, the method

```
(boolean, Any)
  next;
```

can be used to repeatedly retrieve the next object. The value returned by `next` is a tuple with a `boolean` and an object. If the `boolean` is `TRUE`, the second element of the tuple contains the object retrieved. Otherwise, if it is `FALSE`, the end of the collection has been reached.

The following program is an example of using this method.

```
int
  main Array argv
{
  Enumerator e = [argv enumerator];
  Any object;

  while ({
           boolean valid;
           (valid, object) = [e next];
           valid;
         })
    [[[stdio out] print ("got one: `", object, '\'')] nl];

  return 0;
}
```

The example uses a compound expression as the condition of the **while** loop. This isn't just to show you this is possible; the tuple returned by **next** can not be used as a boolean condition, and the **boolean** must be repeated to give the compound its **boolean** value.

If the declaration of **valid** were not part of this compound (but put outside the **while** loop), the condition could be written as

```
while ({(valid, object) = [e next]; valid;})
  ...
```

which is the common notation.

In addition to providing multi-valued returns, and a compact method argument notation, tuples provide a sometimes desirable feature of simultaneous assignment. The most obvious application thereof is that of swapping the values of two variables:

```
(a, b) = (b, a);
```

3.2 Return values

Up to now, returning from a method was always written using **return**. This sets the value to be returned and terminates execution of the method. Sometimes it is desirable to set the return value but not immediately return from the method. Such constructions are common in languages which only provide the **return** statement, and resemble the following code snippet:

```
{
  ...
  int result = [self computation];
  [self deallocateResources];
  return result;
}
```

Using the *return assignment*, the return value of a method can be set, without causing immediate termination. A return assignment is written as an assignment with an empty left-hand side. The example then becomes:

```
{
  ...
  = [self computation];
  [self deallocateResources];
}
```

which is much cleaner. It is customary to only use **return** when immediate termination of the method is necessary. The return assignment is otherwise preferred.

When a method does not assign a return value, either through **return** or a return assignment, the value returned by the method will be the default value of the return type. Thus, the following two method definitions are equal:

```
boolean
  constantp
{
  boolean v;
  = v;
}

boolean
  constantp
{
}
```

When a method returns a tuple the return value returned must be set atomically, i.e. it is only possible to set all values in the tuple at once. There are situation where this is undesirable, and a solution to this is provided by *named return values*.

In the following example, the return type is followed by a tuple of identifiers `(valid, object)`. Each identifier denotes a local variable in the method, which is declared implicitly, and which is handled as a normal local variable. Upon return from the method, the value returned will be the tuple `(valid, object)`. Intermediate assignments to any of these variables will have the expected result of affecting the return value. Also, normal `return` and return assignment still operate as expected, i.e. they will affect the value of these variables.

```
(bool, Any) (valid, object)
  next
{
  valid = [self haveMoreObjects];
  if (valid)
    object = [self getNextObject];
}
```

If one of the return value names is the name of an argument, be it the implicit receiver object `self` or the message selector `cmd` or a normal argument, that part of the return value will correspond to the value of the argument. The creation of a local variable is omitted in this case. The following method for example simply returns the value passed to it:

```
int (value)
  echo int value
{
}
```

Note that the return value name is enclosed in parentheses, i.e. a singleton tuple. Return value names must always be a tuple, otherwise the compiler can't discern between the name and the return value or the first method name part.

A good reason to name the return values is that any documentation or comments about the method can simply refer to parts of the return value by name.

Chapter 4

Objects

We now come to the most important subject of an object oriented programming language: how objects are constructed. We will use the `Counter` class first described in chapter 1 as the example along which to explain the object basics. From the informal description in that chapter we know the following:

1. each `Counter` object maintains information about its current value, and

2. it responds to the `nextValue` message by returning the next value, thereby updating its current value.

This description only contains statements about the `Counter` objects, i.e. instances of the `Counter` class. Therefore the class definition can be empty, for now.

```
implementation class Counter end;
```

The `Counter` instances maintain a current value, and respond to the `nextValue` method. This is written thus:

```
implementation instance Counter
{
  int current_value;
}
```

```
int
  nextValue
{
  current_value += 1;
  = current_value;
}

end;
```

The definition of the `Counter` instance starts with `implementation instance Counter` and ends with `end;`. Within this definition, between the first pair of braces an *instance variable* is defined. Instance variables form the state maintained by each instance of the current class: every `Counter` instance will have its own value of the `current_value`. This variable has the type `int`.

Following the instance variable declaration, the `nextValue` method is defined. This method has no arguments and returns an `int`. In the body of the method, the `current_value` instance variable of the *current instance* is incremented by 1, and the resulting value is returned. The current instance is the receiver of the message as a result of which the current method was invoked. Thus, in the following example

```
Counter count = ...;
int i = [count nextValue];
```

the `count` variable denotes a `Counter` instance; in the second line, the `nextValue` message is sent to it, and the `nextValue` method will be invoked, with the object we know as `count` as the receiver.

4.1 Inheritance

We now have a class definition, but no means yet to create instances. The language does not provide a mechanism to create instances; instead, this functionality is provided by the library. The TOM standard library contains a class `State` that provides the following method:

```
instance (id)
  alloc;
```

This class method returns a new instance of the receiving class. Thus, invoking

```
x = [State alloc];
```

will return a new instance of the `State` class. "How does this help us to create `Counter` objects?" you ask. By indicating that a `Counter` object is also a `State` object, which will cause the `Counter` objects to behave as `State` objects, and similar, that the `Counter` class behaves as the `State` class. This is called *inheritance*: every method defined by `State` is not only applicable to `State`, but also to `Counter`. To indicate that our `Counter` inherits from `State`, the class definition is changed to:

```
implementation class
Counter: State

end;
```

Now any method or instance variable defined for the `State` class (or instance) now also applies to the `Counter` class (or instance). To denote the relationship between the two classes, `State` is called a *superclass* of `Counter`, and `Counter` is a *subclass* of `State`.

The methods inherited from `State` include the `alloc` method and we can now send `alloc` messages to the `Counter` class to create new `Counter` instances:

```
Counter count1 = [Counter alloc];
```

The return type of the `alloc` method is `instance (id)`. The `id` type denotes the type of the receiving object, as seen by the caller. In the example, the receiver is the `Counter` class object, and `id` denotes the `Counter` class. The `instance ()` modifies the type between the parentheses to indicate the instance of that type. The type of object returned by the `alloc` method in this example thus is 'instance of the `Counter` class', written as `Counter`, which is exactly the type of the `count1` variable to which the value returned is assigned.

4.2 Method overriding

Suppose we need a `TwoCounter` object that increments its value by 2 instead of 1 each time `nextValue` is invoked. We could write a `TwoCounter` class from scratch en end up with a class very similar to `Counter`, but that wastes the effort we put in developing the `Counter` class and duplicates that effort in creating the `TwoCounter` class. The severity of this problem increases with the complexity of the classes and the effort needed to develop[1] them.

Luckily, just like `Counter` inherits from `State`, `TwoCounter` can inherit from `Counter`. Since we need different behaviour for the `nextValue` message, the `TwoCounter` can provide its own `nextValue` method, overriding the method provided by `Counter`.

```
implementation class TwoCounter: Counter end;


implementation instance
TwoCounter

redefine int
  nextValue
{
  current_value += 2;
  = current_value;
}

end;
```

When `nextValue` is sent to a `TwoCounter` instance, the object will add 2 to its `current_value` and return the result. Apart from this method, a `TwoCounter` behaves exactly the same as a `Counter`.

4.3 Messaging `super`

In the previous section, the `nextValue` method in the `TwoCounter` class was a complete rewrite of the original method. If the method being overridden performs quite a heavy task, it is a waste to have to fully rewrite or copy it, when all that is needed is a slight modification before or afterwards. In the example, all that is needed by the `nextValue` of `TwoCounter` is an increase of the `current_value` before the code of the original method.

The original method can still be invoked, by messaging the special receiver `super`, as is show in this example:

[1] And debug, test, document, etc.

```
redefine int
  nextValue
{
  current_value++;
  = [super nextValue];
}
```

The effect of messaging **super** is that the method invoked will be the method defined or inherited by the superclass of the current class. The value of **self** within that method will be the same as in the current method, thus messaging super is like messaging **self**, with the only difference being an indication which class is to provide the method implementation.

Messaging super need not just concern the method overridden by the current method. Any method can be invoked, though that is highly unusual[2].

The **redefine** qualifier shown in the example method definitions in this section are actually optional. The compiler can be directed to issue a warning when a **redefine** is omitted, but due to reasons to become clear later, the presence of **redefine** can not be required.

4.4 Object initialization

An object returned by **alloc** is in a known state: each instance variable of the object has the default value of its type[3]. This state however is not necessarily a meaningful state. For example, if the specification of the **Counter** objects included that the first value normally returned by an instance is 10, the proper initial value of the **current_value** would be 9. Therefore, every object must be initialized after allocation. The default initialization method has no arguments; it is defined by **State** as

```
id
  init
{
  = self;
}
```

[2]Except in the case of initializers, where the designated initializer of the subclass invokes the designated initializer of the superclass. Though actually different these methods are *conceptually* identical.

[3]This excludes those instance variables introduced by `State`, most notably `isa`, which is a reference to the object's class.

The conventional way of creating a new instance of a class is by invoking `alloc` and `init` in a single expression.

```
MyClass x = [[MyClass alloc] init];
```

If `Counter` objects would indeed return 10 as the first value returned from `nextValue`, the initialization method would look like this:

```
id
  init
{
  next_value = 9;
  = [super init];
}
```

Often, initialization of an object needs one or more arguments. An example of this is an initializer for the `Counter` class where the first value returned by `nextValue` can be specified.

```
id
  initWithValue int value
{
  next_value = value - 1;
  = [super init];
}
```

In case of multiple initializers, usually, one is the designated initializer and the other initializers can be implemented by invoking it. For example, if `initWithValue` were the designated initializer of a `Counter`, the `init` method could be implemented like this:

```
id
  init
{
  = [self initWithValue 10];
}
```

An advantage of this setup is that subclasses only need to override a single initializer, when necessary, instead of all or any number of them.

Always having to invoke two methods (`alloc` and some initializer) just to create a new object can become a burden. For this reason, classes can provide, through inheritance or by implementation, one or more *allocators*, which pack the allocation and initialization into a single method. For example, `State` provides `new` as the default allocator: `new` allocates a new object and invokes the default initializer.

```
instance (id)
  new
{
  = [[self alloc] init];
}
```

Using the allocator, objects can be created easier, though not faster since it involves an extra method call.

```
MyClass obj = [MyClass new];
```

Just like a class can have a different designated initializer than its superclass, it can also have a different designated allocator. The `Number` class for instance, provides an allocator `with` for easy allocation:

```
Number one = [IntNumber with 1];
```

4.5 Polymorphism

A subclass understands the same messages as its superclass since every method defined by the superclass is inherited by the subclass if it doesn't provide its own method definition for the particular message. In the case of `Counter` and `TwoCounter` this means that any place where an object is handled as if it is a `Counter` instance, a `TwoCounter` instance can be substituted.

For example, the following method retrieves the `nextValue` from a `Counter` object passed as the argument:

```
int
  getNextValueFrom Counter counter
{
  = [counter nextValue];
}
```

When this method is passed a `TwoCounter` object instead of the expected `Counter`, sending the `nextValue` method is still valid. In fact, sending *any* message understood by a `Counter` will be valid, since the `TwoCounter` is a subclass of the expected `Counter` class. This observation is universally applicable: any time a certain class is expected, passing a subclass is equally valid.

Now, what happens when the following code is executed?

```
Counter c2 = [TwoCounter alloc];
int i = [c2 nextValue];
```

The variable `c2` has type `Counter`, but actually references a `TwoCounter` object. When the `nextValue` message is sent, which method is actually invoked: the one defined by `Counter`, because that is the type of `c2`, or the one defined by `TwoCounter`, because that is the actual class of the receiver? The answers is that the actual class of the receiver, and not the caller's idea of the receiver's type, decides which method is invoked. This is called *polymorphism*: the method invoked depends only on the receiver.

When looking back you'll notice that at a lot of places in the preceding sections, the polymorphism was already used, without mentioning it. Yet, intuitively, the meaning was always obvious.

4.6 A class is not atomic

A class defines the behaviour of its instances, and the state that they carry in support of that behaviour. You can use this class as-is, or subclass it to provide more specialized behaviour. This is where the story ends with most object oriented programming languages, but not with TOM. In circumstances where you have no control over a class, for instance because it is part of a vendor-supplied library, not being able to amend it to your needs might just mean that you can not use the class or, even worse, the whole library.

TOM classes are not atomic. A class can be modified: you can add methods, instances variables and superclasses, without modification to the original source, which is essential in case you do not have access to it. Modification can take place at compile, link, and run-time[4]. Methods can also be replaced, for example to fix erroneous behaviour of the original implementation.

Suppose the `Counter` class we have used as an example throughout this chapter is supplied to us without the source code. It doesn't fully suit our needs, but we do not want to waste any effort rewriting it and we have no control over some locations in the code where `Counter` instances are allocated, making subclassing not an option. We can write an *extension* of `Counter` to make it suitable for our purpose.

[4] A limitation of the current runtime library is that adding instance variables at run time to a class of which instances have been allocated is not possible.

Suppose the needed extra functionality is a `current_value` method which returns the current value of the `Counter`. We can supply this functionality in the following extension:

```
implementation instance
Counter extension fix

int
  current_value
{
  = current_value;
}

end;
```

4.7 Multiple inheritance